CS2500 Exam 2 Rubric — Fall 2013

12 POINTS

Problem 1 (i) Identify the correct/incorrect data definitions below and explain in fewer than 15 words why they are correct/incorrect.

1.
```
(define-struct snake (head tail))
;; A Snake is one of:
;; -- (make-snake Posn Snake)
```

2.
```
(define-struct 3tree (left middle right))
;; A TTree is one of:
;; -- Symbol
;; -- (make-3tree TTree TTree TTree)
```

(1)

$\underset{1}{\text{Snake}}\ \underset{2}{\text{is}}\ \underset{3}{\textit{ill-defined}}$ because $\underset{4}{\text{it}}\ \underset{5}{\text{is}}\ \underset{6}{\text{impossible}}\ \underset{7}{\text{to}}\ \underset{8}{\text{generate}}\ \underset{9}{\text{examples}}$.

(2)

$\underset{1}{\text{TTree}}\ \underset{2}{\text{is}}\ \underset{3}{\text{correct}}\ \underset{4}{\text{because}}\ \underset{5}{\text{it}}\ \underset{6}{\text{uses}}\ \underset{7}{\text{built-in}}\ \underset{8}{\text{forms}}\ \underset{9}{\text{of}}\ \underset{10}{\text{data}}\ \underset{11}{\text{and}}\ \underset{12}{\text{constructors}}$.

(ii) Data definitions serve two roles: data construction and data recognition.

1. Construct one example per data definition:

```
(define-struct container (name content file))
;; A Container is a
;;     (make-container String [List-of Box] File).
;; A Box is one of:
;;    -- a Container
;;    -- a File
;; A File is a String.
```

2. Name or construct an instance of Fun:

```
;; Fun is a [String Number -> Number]
```

```
;; (1) Containers
(define a-file
  "some file")
(define a-box
  (make-container "hello" (list a-file) "a"))
(define a-container
  (make-container "hello" (list a-box a-file) "a"))

;; (3) Funs
 string-ref
or
(define (f s n) n)
```

Problem 2 Develop templates for these data definitions:

8 POINTS

```
(define-struct leaf (val))
(define-struct straight (next))
(define-struct branch (left right))
;; A [Forest X] is one of:
;;   --  empty
;;   --  (cons [Tree X] [Forest X])
;;
;; A [Tree X] is one of:
;;   --  (make-leaf X)
;;   --  (make-straight [Tree X])
;;   --  (make-branch [Tree X] [Tree X])


;;
;;
;;
;;
(define (template/forest f)
  (cond
    [(empty? f) ...]
    [(cons? f)
     (... (template/tree (first f)) ...
      ... (template/forest (rest f)) ...)]))

(define (template/tree t)
  (cond
    [(leaf? t) (... (leaf-val t) ...)]
    [(straight? t)
     (... (template/tree (straight-next t)) ...)]
    [(branch? t)
     (... (template/tree (branch-left t))
      ... (template/tree (branch-right t)) ...)]))
```

Problem 3 Design a program called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up).

16 POINTS

```
;; [List-of Number] -> NonnegativeNumber
;; compute the average of the non-negative numbers in l up to -999
(check-expect (rainfall '(4 2 -3 -999 2 -999)) 3)
(check-expect (rainfall '(4 2 -3)) 3)
(check-expect (rainfall '(-3)) 0)
(define (rainfall l)
  (average (nn-upto-999 l)))


;; [List-of Number] -> [List-of Number]
;; select the non-negative numbers up to -999 (if it shows up)
(check-expect (nn-upto-999 '(4 2 -3 -999 2 -999)) '(4 2))
(define (nn-upto-999 l)
  (cond
    [(empty? l) '()]
    [else (cond
            [(= (first l) -999) '()]
            [(< (first l) 0) (nn-upto-999 (rest l))]
            [else (cons (first l) (nn-upto-999 (rest l)))])]))


;; [List-of Number] -> [List-of Number]
;; average the numbers in l (if any); else: 0
(check-expect (average '()) 0) ;;
(check-expect (average '(4 2)) 3)
(define (average l)
  (if (empty? l) 0 (/ (foldr + 0 l) (length l))))
```

Problem 4 Here is a data definition for lists that contains at least one item: 14 POINTS

```
;; [LOX1 X] is one of:
;; --  (cons X empty)
;; --  (cons X [LOX1 X])
```

(i) Design the function `join2`, which consumes two pieces of data: `l`, an instance of [`LOX1` X], and `x`, an `X`. It creates another list by inserting `x` between all pairs of neighboring elements in `l` (if there are any).

```
;; [LOX1 X] X -> [List-of X]
;; insert x between any two neighboring items on lox

(check-expect (join2 '("a" "b") ",") '("a" "," "b"))
(check-expect (join2 '("a" "b" "c") ",") '("a" "," "b" "," "c"))

(define (join2 lox x)
  (cond
    [(empty? (rest lox)) lox]
    [else (cons (first lox) (cons x (join2 (rest lox) x)))]))
```

(ii) Design the function `join`, which consumes an arbitrary list `l` of `X`s and an instance of `X`. It inserts the latter between all pairs of neighboring elements in `l` (if there are any).

```
;; [List-of X] X -> [List-of X]
;; insert y between any two neighboring items on lox

(check-expect (join '(a b c) 99) '(a 99 b 99 c))
(check-expect (join '(a) 99) '(a))
(check-expect (join '() 99) '())



(define (join lox y)
  (cond
    [(empty? lox) lox]
    [else #;"now we know lox in [LOX1 X]" (join2 lox y)]))
```

Problem 5 Design `zist`. The function consumes two lists of `Posns`. For each pair of corresponding `Posns` on the two lists, it computes the geometric distance. If there is a `Posn` on one list but no corresponding `Posn` on the other list, it computes the distance to the origin. 8 POINTS

The geometric distance between two `Posns` is computed as follows:

```
;; Posn Posn -> NonnegativeNumber
;; computes the distance between two points
(check-expect (distance (make-posn 1 1) (make-posn 4 5)) 5)
(define (distance p q)
  (sqrt
    (+ (sqr (- (posn-x p) (posn-x q)))
       (sqr (- (posn-y p) (posn-y q))))))


(define ORIGIN (make-posn 0 0))

;; [List-of Posn] [List-of Posn] -> [List-of NonnegativeNumber]
;; compute the list of distances between two corresponding Posn
;; use ORIGIN as default point
(check-expect
  (zist `(,(make-posn 1 1)) `(,(make-posn 4 5))) '(5))
(check-expect
  (zist `(,(make-posn 1 1) ,(make-posn 1 0)) `(,(make-posn 4 5))) '(5 1))
(check-expect
  (zist `(,(make-posn 1 1)) `(,(make-posn 4 5) ,(make-posn 1 0))) '(5 1))
(define (zist k l)
  (cond
    [(and (empty? k) (empty? l)) '()]
    [(and (cons? k) (empty? l))
     (cons (distance (first k) ORIGIN) (zist (rest k) empty))]
    [(and (empty? k) (cons? l))
     (cons (distance (first l) ORIGIN) (zist (rest l) empty))]
    [else
     (cons (distance (first k) (first l)) (zist (rest k) (rest l)))]))
```

Problem 6 Inspect the following data definition:

8 POINTS

```
(define-struct leaf (val))
(define-struct fork (left right))
(define-struct straight (next))
;; An NTree is one of:
;;  --  (make-leaf Number)
;;  --  (make-fork NTree NTree)
;;  --  (make-straight NTree)
```

Design the function `split`. It consumes two pieces of data: `t`, an `NTree`, and `r`, a `Number`. It creates a new `NTree` by turning all `leaf`s in `t` into a `branch` with a `leaf` in each field:

- If `r` is smaller than the `val` field, `r` goes into the new left `leaf` and the `val` field becomes the right sub-tree.
- If `r` is greater than the `val` field, `r` goes into the new right `leaf` and the `val` field becomes the left sub-tree.

You may assume that `r` is not equal to any `Number` in `t`.

```
;; NTree Number -> NTree
;; grow tree by splitting all leafs


(check-expect
 (split (make-leaf 1) 0) (make-fork (make-leaf 0) (make-leaf 1)))
(check-expect
  (split (make-straight (make-leaf 1)) 0)
  (make-straight (make-fork (make-leaf 0) (make-leaf 1))))
(check-expect
  (split (make-straight (make-fork (make-leaf 0) (make-leaf 2))) 1)
  (make-straight
    (make-fork
      (make-fork (make-leaf 0) (make-leaf 1))
      (make-fork (make-leaf 1) (make-leaf 2)))))
```

```
(define (split t r)
  (cond
    [(leaf? t)
     (if (< (leaf-val t) r)
         (make-fork t (make-leaf r))
         (make-fork (make-leaf r) t))]
    [(fork? t)
     (make-fork (split (fork-left t) r) (split (fork-right t) r))]
    [else
     (make-straight (split (straight-next t) r))]))
```